Semantic Subtyping with an SMT Solver

Cătălin Hrițcu, Saarland University, Saarbrücken, Germany

Joint work with Andy Gordon, Gavin Bierman, and Dave Langworthy (all from Microsoft)

Refinement Types + Type-test

- Microsoft's "M" language has two very interesting features
  - General refinement types $(x : T \ \textbf{where} \ e)$
    - The subtype containing all values that satisfy a Boolean expression
  - Dynamic type tests $e \ \textbf{in} \ T$
    - Boolean expression testing whether expression belongs to a type
- Each useful in isolation
  - Refinement types can express pre-/post-conditions + invariants
- Combination very powerful

The Big Promise

- Union types $T \mid U \triangleq (x : \mathsf{Any}\ \textbf{where}\ (x\ \textbf{in}\ T) \mid\mid (x\ \textbf{in}\ U))$
- Intersection types $T \,\&\, U \triangleq (x : \mathsf{Any}\ \textbf{where}\ (x\ \textbf{in}\ T)\ \&\&\ (x\ \textbf{in}\ U))$
- Negation types $!T \triangleq (x : \mathsf{Any}\ \textbf{where}\ !(x\ \textbf{in}\ T))$
- Sum types $T + U \triangleq ([\textbf{true}] * T) \mid ([\textbf{false}] * U)$
- Dependent pairs $(\Sigma x : T. U) \triangleq (p : T * \mathsf{Any}\ \textbf{where let}\ x = p.\mathsf{fst}\ \textbf{in}\ (p.\mathsf{snd}\ \textbf{in}\ U))$
- Recursive types $\mu X.T \triangleq (y : \mathsf{Any}\ \textbf{where}\ P(y))$
  $P(y : \mathsf{Any}) : \mathsf{Logical}\ \{y\ \textbf{in}\ T[(y : \mathsf{Any}\ \textbf{where}\ P(y))/X]\}$
- Algebraic datatypes $\mathsf{List}_T = \mu X.((T * X) + \mathsf{unit})$
- **Expresivity:** very simple core calculus that can encode:
  all these typing idioms (and more) + all essential features of M

Microsoft Research Cambridge

The Big Challenge

- Q: Is $(y.\ell) + 42$ well-typed (safe) when y has type …?

  y : Text **NO!** y is a string

  y : Any **NO!** y could be a string

  $y : \{\ell : \text{Integer}\}$ **YES!** y is a record (entity) with (at least) integer field ℓ

  $y : (x : \text{Any } \mathbf{where}\ x\ \mathbf{in}\ \{\ell : \text{Integer}\})$ **YES!** the same as above

  $y : (x : \{\ell : \text{Any}\}\ \mathbf{where}\ x.\ell\ \mathbf{in}\ \text{Integer})$ **YES!** the same as above

  $y : \{\ell : (x : \text{Any } \mathbf{where}\ x == 7)\}$ **YES!** $y.\ell$ is always the integer 7

  $y : (x : \text{Any } \mathbf{where\ false})$ **YES!** vacuously

  $y : (x : \{\ell : \text{Any}\}\ \mathbf{where}\ !(x.\ell\ \mathbf{in}\ \text{Text})\ \&\&\ !(x.\ell\ \mathbf{in}\ \text{Logical})\ \&\&\ \ldots)$ **YES!**

The Big Challenge

- Q: Is $(y.\ell) + 42$ well-ty

y : Text **NO!** y is a string

y : Any **NO!** y could be a

$y : \{\ell : \text{Integer}\}$ **YES!** y is

$y : (x : \text{Any } \mathbf{where}\ x\ \mathbf{in}\ \{\ell$

$y : (x : \{\ell : \text{Any}\}\ \mathbf{where}\ x.$

$y : \{\ell : (x : \text{Any } \mathbf{where}\ x =$

$y : (x : \text{Any } \mathbf{where\ false})$

$y : (x : \{\ell : \text{Any}\}\ \mathbf{where}\ !(x.\ell\ \mathbf{in}\ \text{Text})\ \&\&\ !(x.\ell\ \mathbf{in}\ \text{Logical})\ \&\&\ \ldots)$ **YES!**

Expressivity

Statically type-checking even toy examples becomes hard in this setting.

Type information can be hidden deep inside arbitrarily complicated refinements

Such “strange” types (just much larger) do appear in practice: e.g. all our encodings

Observation: it's all about subtyping!

- **But structural subtyping simply can't handle this**

$\text{Text} <: \{\ell : \text{Integer}\}$

$\text{Any} <: \{\ell : \text{Integer}\}$

$\{\ell : \text{Integer}\} <: \{\ell : \text{Integer}\}$

$(x : \text{Any } \textbf{where } x \textbf{ in } \{\ell : \text{Integer}\}) <: \{\ell : \text{Integer}\}$

$(x : \{\ell : \text{Any}\} \textbf{ where } x.l \textbf{ in } \text{Integer}) <: \{\ell : \text{Integer}\}$

$\{\ell : (x : \text{Any } \textbf{where } x == 7)\} <: \{\ell : \text{Integer}\}$

$(x : \text{Any } \textbf{where false}) <: \{\ell : \text{Integer}\}$

$(x : \{\ell : \text{Any}\} \textbf{ where } !(x.\ell \textbf{ in} \text{Text}) \ \&\& \ !(x.\ell \textbf{ in } \text{Logical}) \ \&\& \ \ldots) <: \{\ell : \text{Integer}\}$

Our Solution

- We use **semantic subtyping**
  - Types are interpreted as FOL formulas $\mathbf{F}[\![T]\!](y)$
    - For instance:

$$\mathbf{F}[\![(x : \mathsf{Any}\ \mathbf{where}\ \mathbf{false})]\!](y) = \mathbf{true} \wedge \mathbf{false}$$

$$\mathbf{F}[\![\{\ell : \mathsf{Integer}\}]\!](y) = \mathsf{is\_E}(y) \wedge \mathsf{v\_has\_field}(\ell, y) \wedge \mathsf{In\_Integer}(\mathsf{v\_dot}(\ell, y))$$

  - Subtyping is defined logical implication

$$T <: U \ \ \mathit{iff}\ \ \models \forall y.\, \mathbf{F}[\![T]\!](y) \implies \mathbf{F}[\![U]\!](y)$$

    - So clearly:

$$(x : \mathsf{Any}\ \mathbf{where}\ \mathbf{false}) <: \{\ell : \mathsf{Integer}\}$$

  - We use an SMT solver to discharge such proof obligations

DMINOR: THE CORE OF M

Dminor Calculus

$S,T,U ::=$	type
Any	the top type
$\mathsf{Integer} \mid \mathsf{Text} \mid \mathsf{Logical}$	scalar type
$T*$	collection type
$\{\ell : T\}$	record/entity type (single; open)
$(x : T \textbf{ where } e)$	refinement type
$e ::=$	expression
$x \mid c$	variable or constant
$\oplus(e_1, \ldots, e_n)$	operator application
$e_1 ? e_2 : e_3$	conditional
$\textbf{let } x = e_1 \textbf{ in } e_2$	let-expression
$e \textbf{ in } T$	dynamic type-test
$e : T$	type ascription
$\{\ell_i \Rightarrow e_i \,^{i \in 1..n}\}$	record/entity
$e.\ell$	field selection
$\{v_1, \ldots, v_n\}$	collection (multiset; unordered)
$e_1 :: e_2$	adding element e_1 to collection e_2
$\textbf{from } x \textbf{ in } e_1 \textbf{ let } y = e_2 \textbf{ accumulate } e_3$	fold over collection
$f(e_1, \ldots, e_n)$	function application

Accumulate example

$$\mathsf{NullableInt} \triangleq \mathsf{Integer} \mid [\mathbf{null}]$$

$$\begin{array}{l} \mathsf{removeNulls}(\mathsf{xs} : \mathsf{NullableInt}{*}) : \mathsf{Integer}{*}\ \{ \\ \qquad \mathbf{from}\ \mathsf{x}\ \mathbf{in}\ \mathsf{xs} \\ \qquad \mathbf{let}\ \mathsf{a} = \{\} : \mathsf{Integer}{*} \\ \qquad \mathbf{accumulate}\ (\mathsf{x}{!}{=}\mathbf{null})\ ?\ (\mathsf{x} :: \mathsf{a}) : \mathsf{a} \\ \} \end{array}$$

$$\mathsf{removeNulls}(\{1, \mathbf{null}, 42, \mathbf{null}\} \to^{*} \{1, 42\} = \{42, 1\}$$

Purity

- Dminor side-effects: non-termination and non-determinism
- Expressions in refinement types have to be "pure" (and Logical)

$$\frac{E, x : T \vdash e : \text{Logical} \qquad e \text{ pure}}{E \vdash (x : T \textbf{ where } e)}$$

- Pure expressions are terminating and have unique normal form
- Checking expression purity:
  – $f(e_1, \ldots, e_n)$ is pure only if f terminates on all inputs
    - Syntactic termination condition enforces that recursive calls are made only on structurally smaller arguments
  – from x in e_1 let y = e_2 accumulate e_3 should converge ("$\lambda x\ y.\ e_3$" needs to be associative and commutative)

Singleton + "OK" types

- We have seen encodings for: union, intersection, negation, sum, dependent pair, recursive, algebraic types

- Singleton types

$$[e : T] \triangleq \begin{cases} (x : T \ \textbf{where} \ x == e) & \text{if } e \text{ pure} \\ T & \text{otherwise} \end{cases}$$

- "OK" types

$$\mathsf{Ok}(e) \triangleq \begin{cases} (x : \mathsf{Any} \ \textbf{where} \ e) & \text{if } e \text{ pure} \\ \mathsf{Any} & \text{otherwise} \end{cases}$$

Declarative type system

(Exp Subsum)
$$\frac{E \vdash e : T \quad E \vdash T <: T'}{E \vdash e : T'}$$

(Exp Singleton)
$$\frac{E \vdash e : T}{E \vdash e : [e : T]}$$

(Exp Test)
$$\frac{E \vdash e : \mathsf{Any} \quad E \vdash T}{E \vdash e \ \mathbf{in}\ T : \mathsf{Logical}}$$

(Exp Cond)
$$\frac{E \vdash e_1 : \mathsf{Logical} \quad E, \_ : \mathsf{Ok}(e_1) \vdash e_2 : T \quad E, \_ : \mathsf{Ok}(!e_1) \vdash e_3 : T}{E \vdash (e_1 ? e_2 : e_3) : T}$$

(Exp Dot)
$$\frac{E \vdash e : \{\ell : T\}}{E \vdash e.\ell : T}$$

- **Sound**: well-typed expressions don't cause typing errors
- **Declarative**: uses magic non-determinism; specifies what, not how

Declarative type system

(Exp Singular Subsum)

$$\frac{E \vdash e : T \quad E \vdash [e : T] <: T'}{E \vdash e : T'}$$

(Exp Test)

$$\frac{E \vdash e : \mathsf{Any} \quad E \vdash T}{E \vdash e \textbf{ in } T : \mathsf{Logical}}$$

(Exp Cond)

$$\frac{E \vdash e_1 : \mathsf{Logical} \quad E, \_ : \mathsf{Ok}(e_1) \vdash e_2 : T \quad E, \_ : \mathsf{Ok}(!e_1) \vdash e_3 : T}{E \vdash (e_1 ? e_2 : e_3) : T}$$

(Exp Dot)

$$\frac{E \vdash e : \{\ell : T\}}{E \vdash e.\ell : T}$$

- **Sound**: well-typed expressions don't cause typing errors
- **Declarative**: uses magic non-determinism; specifies what, not how

Bidirectional typing rules

- Two additional algorithmic judgments
  - Type synthesis: $E \vdash e \rightarrow T$ (computes the “strongest” type for *e*)
  - Type checking: $E \vdash e \leftarrow T$ (tests whether *e* has type *T*)

(Swap)
$$\frac{E \vdash e \rightarrow T \quad E \vdash [e : T] <: T'}{E \vdash e \leftarrow T'}$$

(Synth Test)
$$\frac{E \vdash e \leftarrow \mathsf{Any} \quad E \vdash T}{E \vdash e \ \mathbf{in}\ T \rightarrow \mathsf{Logical}}$$

(Check Dot)
$$\frac{E \vdash e \leftarrow \{\ell : T\}}{E \vdash e.\ell \leftarrow T}$$

- Expressivity strikes [us] again!

$$y : (x : \{\ell : \mathsf{Any}\}\ \mathbf{where}\ !(x.\ell\ \mathbf{in}\ \mathsf{Text})) \vdash y.\ell \rightarrow \quad ???$$

Bidirectional typing rules

- Two additional algorithmic judgments
  - Type synthesis: $E \vdash e \rightarrow T$ (computes the "strongest" type for *e*)
  - Type checking: $E \vdash e \leftarrow T$ (tests whether *e* has type *T*)

(Swap)
$$\frac{E \vdash e \rightarrow T \quad E \vdash [e : T] <: T'}{E \vdash e \leftarrow T'}$$

(Synth Test)
$$\frac{E \vdash e \leftarrow \mathsf{Any} \quad E \vdash T}{E \vdash e \ \mathbf{in}\ T \rightarrow \mathsf{Logical}}$$

(Check Dot)
$$\frac{E \vdash e \leftarrow \{\ell : T\}}{E \vdash e.\ell \leftarrow T}$$

(Synth Dot)
$$\frac{E \vdash e \rightarrow T \quad norm(T) = D \quad D.\ell \rightsquigarrow U}{E \vdash e.\ell \rightarrow U}$$

- Expressivity strikes [us] again!

$$y : (x : \{\ell : \mathsf{Any}\}\ \mathbf{where}\ !(x.\ell\ \mathbf{in}\ \mathsf{Text})) \vdash y.\ell \rightarrow \ !\mathsf{Text}$$

Semantic subtyping

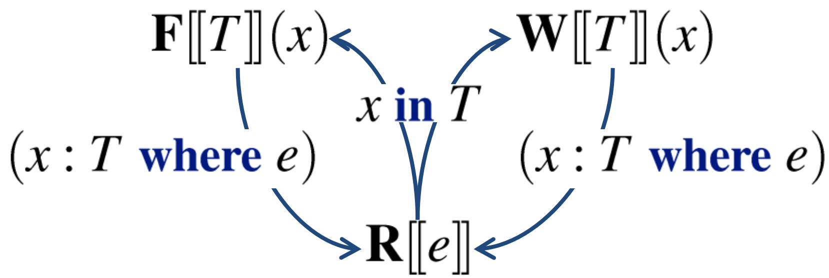
- Types interpreted as FOL formulas $\mathbf{F}[\![T]\!](x)$
- Subtyping is just implication between interpretations

$$\text{(Subtype)}\quad \frac{E \vdash T \quad E \vdash T' \quad \models (\mathbf{F}[\![E]\!] \implies (\forall x.\ \mathbf{F}[\![T]\!](x) \implies \mathbf{F}[\![T']\!](x))}{E \vdash T <: T'}$$

- These formulas interpreted in specific FOL model
  – We formalized this model in Coq (once and for all, ~2000LOC)
    - FOL sort $\longrightarrow$ Coq type
    - FOL function symbol $\longrightarrow$ Coq function
  – We feed properties of the model as “axioms” to the SMT solver

Logical Semantics

- We define three mutually recursive translations



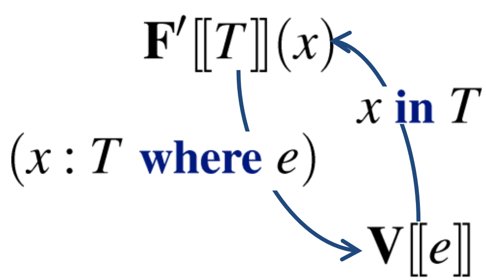
  - $\mathbf{F}[\![T]\!](x)$ – formula: is value x in type *T*?
  - $\mathbf{R}[\![e]\!]$ – term: the result of evaluating pure *e* (a value or Error)
  - $\mathbf{W}[\![T]\!](x)$ – formula: does checking whether *x* is in *T* go wrong?
- This error-tracking semantics is fully abstract, but complicated

Optimized Logical Semantics

- **Observation**: we only care about well-formed types and well-typed (+ pure) expressions

$$\text{(Subtype)}\quad \frac{E \vdash T \quad E \vdash T' \quad \models (\mathbf{F}'[\![E]\!] \implies (\forall x.\ \mathbf{F}'[\![T]\!](x) \implies \mathbf{F}'[\![T']\!](x))}{E \vdash T <: T'}$$

- We don't need to track errors, which simplifies things a lot



Optimized Semantics of Types: $\mathbf{F}'[\![T]\!](t)$

$$\mathbf{F}'[\![\mathsf{Any}]\!](v) = \textbf{true}$$
$$\mathbf{F}'[\![\mathsf{Integer}]\!](v) = \mathsf{In\_Integer}(v)$$
$$\mathbf{F}'[\![\mathsf{Text}]\!](v) = \mathsf{In\_Text}(v)$$
$$\mathbf{F}'[\![\mathsf{Logical}]\!](v) = \mathsf{In\_Logical}(v)$$
$$\mathbf{F}'[\![\{\ell : T\}]\!](v) = \mathsf{is\_E}(v) \wedge \mathsf{v\_has\_field}(\ell, v) \wedge \mathbf{F}'[\![T]\!](\mathsf{v\_dot}(v, \ell))$$
$$\mathbf{F}'[\![T*]\!](v) = \mathsf{is\_C}(v) \wedge (\forall x.\mathsf{v\_mem}(x, v) \Rightarrow \mathbf{F}'[\![T]\!](x)) \quad x \notin fv(T, v)$$

$$\mathbf{F}'[\![(x : T \ \textbf{where}\ e)]\!](v) = \mathbf{F}'[\![T]\!](v) \wedge \textbf{let}\ x = v\ \textbf{in}\ \mathbf{V}[\![e]\!] = \textbf{true}$$

Optimized Semantics of Pure Typed Expressions: $\mathbf{V}[\![e]\!]$

$$\mathbf{V}[\![\oplus(e_1, \ldots, e_n)]\!] = \mathsf{O}_\oplus(\mathbf{V}[\![e_1]\!], \ldots, \mathbf{V}[\![e_n]\!])$$
$$\mathbf{V}[\![e_1 ? e_2 : e_3]\!] = (\textbf{if}\ x = \textbf{true then}\ \mathbf{V}[\![e_2]\!]\ \textbf{else}\ \mathbf{V}[\![e_3]\!])$$
$$\mathbf{V}[\![\textbf{let}\ x = e_1\ \textbf{in}\ e_2]\!] = \textbf{let}\ x = \mathbf{V}[\![e_1]\!]\ \textbf{in}\ \mathbf{V}[\![e_2]\!]$$
$$\mathbf{V}[\![e\ \textbf{in}\ T]\!] = \mathsf{v\_logical}(\mathbf{F}'[\![T]\!](\mathbf{V}[\![e]\!]))$$
$$\mathbf{V}[\![e : T]\!] = \mathbf{V}[\![e]\!]$$
$$\mathbf{V}[\![\{\ell_i \Rightarrow e_i\ ^{i \in 1..n}\}]\!] = \{\ell_i \Rightarrow \mathbf{V}[\![e_i]\!]\ ^{i \in 1..n}\}$$
$$\mathbf{V}[\![e.\ell]\!] = \mathsf{v\_dot}(\mathbf{V}[\![e]\!], \ell)$$
$$\mathbf{V}[\![\{v_1, \ldots, v_n\}]\!] = \{v_1, \ldots, v_n\}$$
$$\mathbf{V}[\![e_1 :: e_2]\!] = \mathsf{v\_add}(\mathbf{V}[\![e_1]\!], \mathbf{V}[\![e_2]\!])$$
$$\mathbf{V}[\![\textbf{from}\ x\ \textbf{in}\ e_1\ \textbf{let}\ y = e_2\ \textbf{accumulate}\ e_3]\!] = \mathsf{v\_accumulate}((\textbf{fun}\ x\ y \to \mathbf{V}[\![e_3]\!]), \mathbf{V}[\![e_1]\!], \mathbf{V}[\![e_2]\!])$$

Axiomatizing Model in SMT-LIB

- FOL with the following (combination of) standard theories
  - equality + uninterpreted function symbols
  - integer arithmetic (not necessarily linear)
  - algebraic datatypes (Z3-specific extension.to.SMT-LIB)
  - extensional arrays (Z3-specific extension.to.SMT-LIB)
- Main concerns:
  - tradeoff between performance and completeness
  - finding the right quantifier patterns

Implementation

- Around 2700 lines of F#
- Uses Z3 SMT solver (Microsoft Research)
  - Really amazing, gets 1s per proof obligation by default
    - But it usually solves 150 POs/s
  - Much ongoing research on SMT, solvers always getting faster
- Type-checking really fast: 1-3s (tested on 130 files)
- Released under the Microsoft Research License: http://research.microsoft.com/~adg/dminor.html
- Private demos available on request … also see the screencast

Bonuses

1. Precise counterexamples to type-checking

```
foo(n : PosInt, m : PosInt) : PosInt {
  42 + n + m - n * m
}
```

. Can't convert (((42+n)+m)-(n*m)) to type PosInt.
For instance if n->2, m->325 expression evaluates to -281 that does not have type PosInt.

2. Finding elements of types + highlighting empty types

```
(x : Integer where x * x + 42 < 0) + 100 < 42)
```

Empty type

Bonuses

1. Precise counterexamples to type-checking

```
foo(n : PosInt, m : PosInt) : PosInt {
  42 + n + m - n * m
}
```

. Can't convert (((42+n)+m)-(n*m)) to type PosInt.
For instance if n->2, m->325 expression evaluates to -281 that does not have type PosInt.

2. Finding elements of types + highlighting empty types

```
(x : Integer where x * x + 42 < 0) + 100 < 42)
```

Inhabited (e.g. -4)

3. Constraint programming in Dminor   **elementof** *T*

```
GenerateAllGoodMachines(avoid : GoodMachine*) : GoodMachine* {
    let m = elementof (x : GoodMachine where !(x in avoid)) in
    (m == null) ? {} : (m :: (GenerateAllGoodMachines(m :: avoid)))
}
```

C:\Windows\system32\cmd.exe

```
Executing (let g=GenerateAllGoodMachines({}) in (let b=GenerateAllBadMachines({}
) in {GoodMachinesCount=>(g.Count); GoodMachines=>g; BadMachinesCount=>(g.Count)
; BadMachines=>b; })))...

Result of evaluation:          {GoodMachinesCount=>8; GoodMachines=>{{s2=>{port
=>501; name=>"IIS"; }; s1=>{port=>502; name=>"IIS"; }; }, {s2=>{port=>502; name=
>"IIS"; }; s1=>{port=>501; name=>"IIS"; }; }, {s2=>{port=>502; name=>"SQL Server
"; }; s1=>{port=>501; name=>"IIS"; }; }, {s2=>{port=>502; name=>"IIS"; }; s1=>{p
ort=>500; name=>"IIS"; }; }, {s2=>{port=>500; name=>"SQL Server"; }; s1=>{port=>
502; name=>"SQL Server"; }; }, {s2=>{port=>502; name=>"IIS"; }; s1=>{port=>501;
name=>"SQL Server"; }; }, {s2=>{port=>502; name=>"SQL Server"; }; s1=>{port=>501
; name=>"SQL Server"; }; }, {s2=>{port=>502; name=>"SQL Server"; }; s1=>{port=>5
00; name=>"SQL Server"; }; }}; BadMachinesCount=>8; BadMachines=>{{s2=>{port=>50
2; name=>"SQL Server"; }; s1=>{port=>502; name=>"IIS"; }; }, {s2=>{port=>501; na
me=>"IIS"; }; s1=>{port=>501; name=>"IIS"; }; }, {s2=>{port=>500; name=>"SQL Ser
ver"; }; s1=>{port=>500; name=>"IIS"; }; }, {s2=>{port=>501; name=>"IIS"; }; s1=
>{port=>501; name=>"SQL Server"; }; }, {s2=>{port=>501; name=>"SQL Server"; }; s
1=>{port=>501; name=>"SQL Server"; }; }, {s2=>{port=>500; name=>"IIS"; }; s1=>{p
ort=>500; name=>"SQL Server"; }; }}; }


C:\Users\hritcu\papers\dminor\microsoft_confidential\dminor-src>
```

3. Constraint programming in Dminor **elementof** *T*

```
GenerateAllGoodMachines(avoid : GoodMachine*) : GoodMachine* {
    let m = elementof (x : GoodMachine where !(x in avoid)) in
    (m == null) ? {} : (m :: (GenerateAllGoodMachines(m :: avoid)))
}
```

Microsoft Research Cambridge

Conclusions

- The first study of [refinement types + dynamic type-case]
- Combination yields great expressivity, but hard to type-check
- Semantic subtyping
  - subtyping is logical implication between the semantics of types
- Type system
  - specified by declarative rules; implemented by bidirectional ones
- Proof obligations discharged using SMT solver (Z3)
  - Bonus: can exploit counterexamples produced by SMT solver
- … and it works: http://research.microsoft.com/~adg/dminor.html

BACKUP SLIDES

Microsoft Research Cambridge

Related Work

			Refinement	Type-test	Subtyping
1983	Nordström/Petersson	**Subset types**	{x:A | B(x)}	no	no
1986	Rushby/Owre/Shankar	**Predicate subtyping**	predicate subtype	no	limited
1989	Cardelli et al	**Modula-3 Report**	no	on references	structural
1991	Pfenning/Freeman	**Refinement types**	refined sorts	no	no
1993	Aiken and Wimmers	**Type inclusion...**	no	no	semantic
1999	Pfenning/Xi	**DML**	{x: General | e}	no	no
1999	Buneman/Pierce	**Unions for SSD**	no	yes, as pattern	structural
2000	Hosoya/Pierce	**XDuce**	no	yes, as pattern	semantic, ad hoc
2006	Flanagan et al	**SAGE**	{x: T | e}	no (but has cast)	structural, SMT
2006	Fisher et al	**PADS**	{x:T | e}	no	structural
2007	Frisch/Castagna	**CDuce**	no	e in T	semantic, ad hoc
2007	Sozeau	**Russell**	{x:T | e}	no	structural
2008	Bhargavan/Fournet/G	**F7/RCF**	{x: T | C} (formula C)	no	structural, SMT
2008	Rondon/Jhala	**Liquid Types**	{x: General | e}	no	structural, SMT
2010	Bierman/G/H/L	**M/Dminor**	{x: T | e}	e in T	semantic, SMT

Other types we can encode

- We already did: union, intersection, negation, singleton, sum, variant, recursive and algebraic types … so what else is left? ☺
- Multi-field entity types

$$\{\ell_i : T_i;\ {}^{i\in 1..n}\} \triangleq \{\ell_1 : T_1\}\ \&\ \ldots\ \&\ \{\ell_n : T_n\}$$

- Closed entity types

$$\textbf{closed}\{\ell_i : T_i;\ {}^{i\in 1..n}\} \triangleq (x : \{\ell_i : T_i;\ {}^{i\in 1..n}\}\ \textbf{where}\ x == \{\ell_i \Rightarrow x.\ell_i,\ {}^{i\in 1..n}\})$$

- Pair types

$$T * U = \textbf{closed}\{\mathsf{fst} : T; \mathsf{snd} : U;\}$$

- Variant types

$$< \ell_1 : T_1; \ldots ; \ell_n : T_n > \triangleq ([\ell_1] * T_1) \mid \ldots \mid ([\ell_n] * T_n)$$

- Self types

$$\textbf{Self}(s : T)U \triangleq (s : T\ \textbf{where}\ s\ \textbf{in}\ U)$$

Formalizing Dminor Model in Coq

- FOL sort ⟶ math set – Coq type

```
Inductive RawValue : Type :=
  | G : General → RawValue
  | E : list (string ∗ RawValue) → RawValue
  | C : list RawValue → RawValue.

Definition Value := {x : RawValue | Normal x}.
```

- FOL function symbol ⟶ total function – Coq function

```
Program Definition v_has_field (s : string) (v : Value) : bool :=
  match TheoryList.assoc eq_str_dec s (out_E v) with
  | Some v ⇒ true
  | None ⇒ false
  end.
```

First-order theories

- Semantics given with respect to a particular logical model
- We use SMT-LIB (+Z3 extensions) to axiomatize this model
- Sorted first-order logic +
  - Integers: build-in sort Int + arithmetic operations

```
:formula (forall (x Int) (= (+ 0 x) x))      ; Z3: valid
```

  - Algebraic datatypes:

```
:datatypes((VList
     Nil
     (Cons (out_Head Value) (out_Tail VList))))
```

  - “Arrays” – updatable functions with finite support

```
:define_sorts ((VArray (array Int Value))     ; C arrays
      (VBag (array Value Int))           ; M collections
      (VMap (array String Value)))         ; M entities
```

Axiomatizing model

- The semantic domain of values

```
:datatypes (
  (Value
    (G (out_G General))              ;; scalar values
    (E (out_E (array String Value)) ;; entities
    (C (out_C (array Value Int)))   ;; collections
  )
```

- Axiomatization of function and predicate symbols

```
:extrafuns((v_tt Value)(v_int Int Value)(O_Sum Value Value
Value))
:assumption (= v_tt (G(G_Logical true)))
:assumption (forall (n Int) (= (v_int n) (G(G_Integer n)))
  :pat { (v_int n) } :pat { (G(G_Integer n)) }
:assumption (forall (i1 Int) (i2 Int)
  (= (O_Sum (v_int i1) (v_int i2)) (v_int (+ i1 i2)))
  :pat { (O_Sum (v_int i1) (v_int i2)) })
```

Axiomatizing collections

- Finiteness of bags

```
:assumption (forall (a (array Value Int))
  (iff (Finite a) (= (default a) 0)))
```

- Only positive indices in bags

```
:assumption (forall (a (array Value Int))
  (iff (Positive a) (forall (v Value) (>= (select a v) 0))
```

- Collections are finite bags with positive indices

```
:assumption (forall (v Value)
  (iff (In_C v)
       (and (is_C v)
            (Finite (out_C v))
            (Positive (out_C v)))))
```

- Collection membership

```
:assumption (forall (v Value) (a (array Value Int))
  (iff (v_mem v (C a)) (> (select a v) 0)))
```

THE END